307307
Generative AI

Introduction to LLMs, Transformers and Context Aware Embeddings

What Are Large Language Models (LLMs)?

- LLMs are neural networks trained on large text corpora to understand and generate human language.
- They learn statistical patterns, grammar, semantics, and world knowledge directly from data.
- Built on the Transformer architecture, which relies on self-attention to capture relationships between all words in a sequence.
- Capable of performing many tasks without task-specific training, such as answering questions, writing code, summarizing text, and translating languages.
- Examples: GPT, PaLM, LLaMA, Claude

How LLMs Are Trained

- **Pretraining:**
  - Models see massive amounts of text and learn by predicting masked or next tokens.
  - This builds general language understanding and knowledge of syntax, semantics, and facts.
- **Fine-tuning:**
  - Models can be adjusted for specific tasks (classification, summarization, translation).
- **Instruction Tuning and RLHF:**
  - Models learn to follow human instructions using curated datasets and human feedback.
  - Ensures outputs are helpful, safe, and aligned with user intent.
- The scale of data and parameters allows the emergence of advanced reasoning and generative abilities.

Pretraining vs. Fine-Tuning

Pretraining Example: A base LLM like GPT is trained on **trillions of tokens** from books, websites, articles, and code.
It learns:

- How sentences are structured
- How logic works
- Broad world knowledge
- General reasoning skills
  This is like teaching the model *all of human language*.

Fine-Tuning Example: A company wants a **customer-support chatbot** for their bank.
They take the pretrained model and fine-tune it using:

- 5,000 chat transcripts
- Bank-specific FAQs
- Compliance rules and preferred tone
  After fine-tuning, the model learns:
- Bank terminology
- Policies for loans, fees, accounts
- Approved response style
- How to avoid giving unauthorized financial advice
- This process takes hours instead of months and costs thousands instead of millions.

How LLMs Work at Inference Time

- Input text is tokenized and converted into embeddings.
- The model processes the input through multiple Transformer layers to build a contextual representation.
- For generation, the model predicts one token at a time, using masked self-attention to avoid “seeing the future.”
- Decoding strategies (greedy, beam search, sampling, top-k, top-p) shape the creativity and quality of the output.
- The model continues generating tokens until an end-of-sequence token or stopping condition is reached.

What LLMs Can Do and Their Limitations

- **Strengths:**
  - Text generation, explanation, summarization, reasoning, coding, translation, question answering.
  - Adaptable to many tasks via prompting, not just training.
- **Limitations:**
  - Can produce incorrect or fabricated information (hallucination).
  - Lack true understanding or grounded world perception.
  - Sensitive to prompt phrasing and context.
  - Require substantial computing resources for training and inference.
- LLMs are powerful tools, but outputs must be interpreted with domain knowledge and verification.

Key Generation Settings in Language Models

Context Window

- The maximum amount of text the model can consider at once.
- A larger context window lets the model remember more of the conversation or document, improving coherence over long inputs.
- Once the limit is reached, older text is no longer considered.

Temperature

- Controls randomness in generation.
- Low temperature (e.g., 0.2) makes the model more focused and deterministic.
- High temperature (e.g., 1.0) makes the model more creative and varied.

Top-N / Top-K Sampling

- Limits the model to choosing from the top K most likely next tokens.
- Lower K makes output more predictable.
- Higher K adds more variety while still avoiding extremely unlikely tokens.
- If you want, I can format this as bullet points for a real slide deck or export it to PowerPoint.

Introduction to Prompt Engineering

Introduction to Prompt Engineering

What is Prompt Engineering?

Prompt engineering is the practice of designing and optimizing input instructions (prompts) to effectively communicate with Large Language Models (LLMs) and achieve desired outputs. It involves crafting queries that guide AI models to produce accurate, relevant, and useful responses without modifying the underlying model parameters.

Core Principles:

- Task specification through natural language
- Leveraging pre-trained knowledge without fine-tuning
- Systematic optimization of input-output mappings
- Context window utilization and management

Categories of Prompting Techniques:

- Zero-shot and Few-shot Learning
- Chain of Thought (CoT) Reasoning
- Instruction-based Prompting
- ReAct and Tool-Augmented Approaches

Zero-Shot and Few-Shot Prompting

Zero-Shot Prompting: You give the model **only instructions**, no examples. The model relies entirely on prior knowledge.
Example Prompt: "Classify this text as Positive, Negative, or Neutral: *'The schedule change was unexpected but manageable.'*"
Output: Neutral

One-Shot Prompting: You give **one explicit example** to show the input–output format.
Example Prompt:
"I want you to classify sentiment.
Example Input: *'Loved the new update!'*
Example Output: Positive
Now classify: *'The service was slow and frustrating.'*"
Output: Negative

Few-Shot Prompting: You give **several examples (2–5)** to establish a clear pattern.
Example Prompt:
"I want you to classify sentiment. Follow the examples:
Input: *'Great value!'* → Positive
Input: *'Not worth the price.'* → Negative
Input: *'Works fine.'* → Neutral
Now classify: *'Exceeded my expectations.'*"
Output: Positive

Chain of Thought (CoT) Prompting

Chain-of-Thought prompting explicitly instructs the model to **break complex problems into intermediate reasoning steps**. This improves accuracy on tasks requiring **multi-step logic, arithmetic, planning, or deduction**.

CoT Variants:

1) Zero-Shot Chain of Thought
You add a reasoning cue (e.g., *"Let's think step by step"*) without providing examples.
Typical performance improvement: **15–25%** on reasoning tasks.
Example Prompt:
"Q: The cafeteria had 23 apples, used 20 for lunch, then bought 6 more.
How many apples do they have? Let's think step by step."

2) One-Shot Chain of Thought Prompting: One-Shot CoT provides **one worked example** with explicit reasoning steps. After the example, you present a new question for the model to solve using the same reasoning style.

Example Prompt (One-Shot CoT):

"I want you to solve the following types of problems using step-by-step reasoning.

Example
Q: Roger has 5 balls. He buys 2 cans with 3 balls each. How many balls does he have?
A: Roger starts with 5 balls. Each can contains 3 balls, so 2 cans = 2 × 3 = 6 balls. Total = 5 + 6 = 11 balls.

Now solve this new problem using the same step-by-step reasoning:

Q: Julia has 7 cats. She adopts 3 pairs of cats. How many cats does she have?
A:"

Advanced Prompting Techniques

Tree of Thoughts (ToT)

- A reasoning framework where the model **explores multiple solution paths**, evaluates them, and backtracks when necessary—similar to a search tree.
  Best for: complex planning, puzzle solving, strategic reasoning, and creative ideation.
- **Simple Prompt Version:**
  "Generate 3 different approaches to solve this problem. For each approach, explain the reasoning, evaluate its strengths and weaknesses, and then choose the best approach."
- **Full ToT Implementation:**
  Uses **multiple model calls** plus an external controller to systematically expand, score, prune, and revisit reasoning branches. (Not achievable with a single prompt alone.)

Constitutional AI Prompting

- Embedding **principles, ethical rules, or behavioral constraints** directly into the prompt to guide safe, consistent responses.
- **Example:**
  "Answer the following question helpfully, harmlessly, and honestly. If the question is unsafe or harmful, provide a safe alternative."

Role-Based Prompting

- Assigning the model a **specific role, expertise, or perspective** to shape tone, depth, and reasoning style.
- **Example:**
  "As an experienced data scientist, explain the concept of overfitting to a beginner using simple language and one illustrative example."

Advanced Prompting Techniques

Definition: ReAct is a prompting framework where the model alternates between:
Thought → Action → Observation,
allowing it to reason step-by-step *and* interact with tools (search, calculators, APIs).
It is ideal for: dynamic problem-solving, research tasks, multi-step queries, and tool-assisted reasoning.
ReAct Format

Thought: model explains its reasoning
Action: the tool operation (search, API call, calculation)
Observation: the result returned by the tool
Repeat until ready
Answer: final solution

Example (ReAct with Tools Enabled)

Thought: I need to find France's GDP to compare it with Germany.
Action: Search["France GDP 2023"]
Observation: $2.78 trillion
Thought: Now get Germany's GDP.
Action: Search["Germany GDP 2023"]
Observation: $4.07 trillion
Answer: Germany's GDP is higher by approximately $1.29 trillion.

How to Implement ReAct

- **With tool-enabled models (e.g., GPT-4 with functions, Claude with tools):**
  The model *actually executes* the actions—running searches, computations, database lookups, etc.
- **Without tools (simulation mode):**
  You can instruct the model to *pretend* to use actions and observations by giving it this structure:
- "Use this format to solve the problem:
  Thought: [your reasoning]
  Action: [what you would search or calculate]
  Observation: [what you would expect that action to return]
  … repeat as needed
  Answer: [final answer]"
- This simulates ReAct, but the model is **not actually searching**—it is imagining the results.

Optional components		
System instructions	Technical or environmental directives that may involve controlling or altering the model's behavior across a set of tasks. For many model APIs, system instructions are specified in a dedicated parameter. System instructions are available in Gemini 2.0 Flash and later models.	You are a coding expert that specializes in rendering code for front-end interfaces. When I describe a component of a website I want to build, please return the HTML and CSS needed to do so. Do not give an explanation for this code. Also offer some UI design suggestions.
Persona	Who or what the model is acting as. Also called "role" or "vision."	You are a math tutor here to help students with their math homework.
Constraints	Restrictions on what the model must adhere to when generating a response, including what the model can and can't do. Also called "guardrails," "boundaries," or "controls."	Don't give the answer to the student directly. Instead, give hints at the next step towards solving the problem. If the student is completely lost, give them the detailed steps to solve the problem.
Tone	The tone of the response. You can also influence the style and tone by specifying a persona. Also called "style," "voice," or "mood."	Respond in a casual and technical manner.
Context	Any information that the model needs to refer to in order to perform the task at hand. Also called "background," "documents," or "input data."	A copy of the student's lesson plans for math.
Few-shot examples	Examples of what the response should look like for a given prompt. Also called "exemplars" or "samples."	`input:` I'm trying to calculate how many golf balls can fit into a box that has a one cubic meter volume. I've converted one cubic meter into cubic centimeters and divided it by the volume of a golf ball in cubic centimeters, but the system says my answer is wrong. `output:` Golf balls are spheres and cannot be packed into a space with perfect efficiency. Your calculations take into account the maximum packing efficiency of spheres.
Reasoning steps	Tell the model to explain its reasoning. This can sometimes improve the model's reasoning capability. Also called "thinking steps."	Explain your reasoning step-by-step.
Response format	The format that you want the response to be in. For example, you can tell the model to output the response in JSON, table, Markdown, paragraph, bulleted list, keywords, elevator pitch, and so on. Also called "structure," "presentation," or "layout."	Format your response in Markdown.
Recap	Concise repeat of the key points of the prompt, especially the constraints and response format, at the end of the prompt.	Don't give away the answer and provide hints instead. Always format your response in Markdown format.
Safeguards	Grounds the questions to the mission of the bot. Also called "safety rules."	N/A

Prompt Engineering Best Practices

Instruction Design Principles:

1) Clarity and Specificity:

Poor: "Summarize this"

Better: "Provide a 3-sentence summary focusing on the main findings and methodology"

2) Output Format Specification:

"Format your response as:

- Main Point: [one sentence]
- Supporting Evidence: [bullet points]
- Conclusion: [one sentence]"

3) Context Window Management:

- Place instructions at beginning and end for long prompts
- Use delimiters for different sections (###, ---, etc.)
- Prioritize relevant information near the query

Prompt Engineering Best Practices

4) Temperature and Parameter Tuning:

- Temperature 0-0.3: Factual, deterministic tasks
- Temperature 0.7-0.9: Creative, diverse outputs
- Top-p sampling: Control output diversity
- Iterative Refinement: Test → Analyze failures → Modify prompt → Repeat

Common Pitfalls to Avoid:

- Ambiguous instructions
- Conflicting requirements
- Assuming implicit knowledge
- Over-engineering simple tasks

Sample prompt template

<OBJECTIVE_AND_PERSONA>

You are a [insert a persona, such as a "math teacher" or "automotive expert"]. Your task is to...

</OBJECTIVE_AND_PERSONA>

<INSTRUCTIONS>

To complete the task, you need to follow these steps:

1.

2.

...

</INSTRUCTIONS>

---------- Optional Components ----------

<CONSTRAINTS>

Dos and don'ts for the following aspects

1. Dos

2. Don'ts

</CONSTRAINTS>

<CONTEXT>

The provided context

</CONTEXT>

More Info and Examples

- https://docs.cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/prompt-design-strategies
- https://github.com/GoogleCloudPlatform/generative-ai/blob/main/gemini/prompts/intro_prompt_design.ipynb

Introduction to the Transformer Architecture

Transformers – Architecture and Principles

Transformer is a neural network architecture based on self-attention mechanism.

It processes all tokens in parallel and models long-range dependencies efficiently.

It was introduced in “Attention Is All You Need” (Vaswani et al., 2017), it became the foundation for most modern language models.

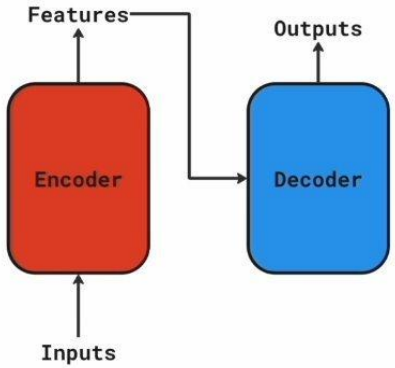
The transformer model consists of two main parts:

1. Encoder:

- The encoder processes the input sequence and generates an encoded representation of it.
- This representation captures the contextual information of the input tokens.

2. Decoder:

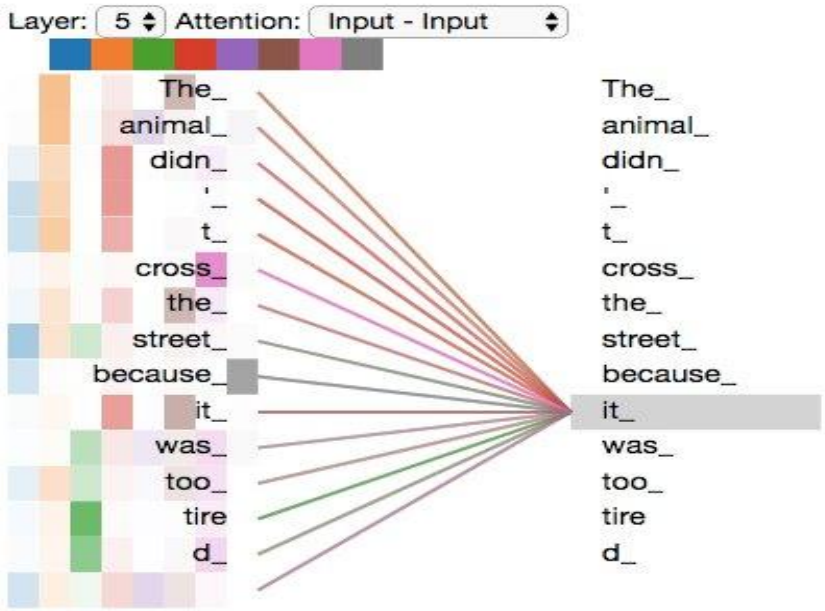
- The decoder takes the encoder's output and generates the final output sequence.
- It does this by predicting one token at a time, using the encoded representations and previously generated tokens.



"Hello world" -> encoder -> knowledge -> -> decoder -> *"Hola mundo"*

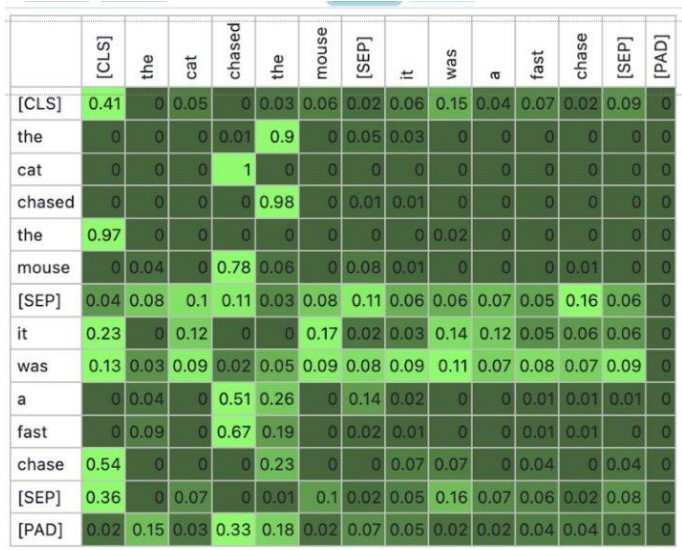
- Encoder: reads and understands text.
- Decoder: generates text one token at a time.

- Attention is a method that lets a language model decide **which words in a sentence matter most to each other**.
- When the model reads a word, it looks at all the other words around it and gives each one a score—basically asking, *"How helpful is this word for understanding the one I'm working on?"*
- These scores are then turned into weights that tell the model **how much to pay attention** to each word.



The heatmap displays attention weights, where each row shows how one token distributes its focus across all other tokens

Brighter cells indicate higher attention values, revealing which tokens are considered most relevant for understanding or generating the current token



	[CLS]	the	cat	chased	the	mouse	[SEP]	it	was	a	fast	chase	[SEP]	[PAD]
[CLS]	0.41	0	0.05	0	0.03	0.06	0.02	0.06	0.15	0.04	0.07	0.02	0.09	0
the	0	0	0	0.01	0.9	0	0.05	0.03	0	0	0	0	0	0
cat	0	0	0	1	0	0	0	0	0	0	0	0	0	0
chased	0	0	0	0	0.98	0	0.01	0.01	0	0	0	0	0	0
the	0.97	0	0	0	0	0	0	0	0.02	0	0	0	0	0
mouse	0	0.04	0	0.78	0.06	0	0.08	0.01	0	0	0	0.01	0	0
[SEP]	0.04	0.08	0.1	0.11	0.03	0.08	0.11	0.06	0.06	0.07	0.05	0.16	0.06	0
it	0.23	0	0.12	0	0	0.17	0.02	0.03	0.14	0.12	0.05	0.06	0.06	0
was	0.13	0.03	0.09	0.02	0.05	0.09	0.08	0.09	0.11	0.07	0.08	0.07	0.09	0
a	0	0.04	0	0.51	0.26	0	0.14	0.02	0	0	0.01	0.01	0.01	0
fast	0	0.09	0	0.67	0.19	0	0.02	0.01	0	0	0.01	0.01	0	0
chase	0.54	0	0	0	0.23	0	0	0.07	0.07	0	0.04	0	0.04	0
[SEP]	0.36	0	0.07	0	0.01	0.1	0.02	0.05	0.16	0.07	0.06	0.02	0.08	0
[PAD]	0.02	0.15	0.03	0.33	0.18	0.02	0.07	0.05	0.02	0.02	0.04	0.04	0.03	0

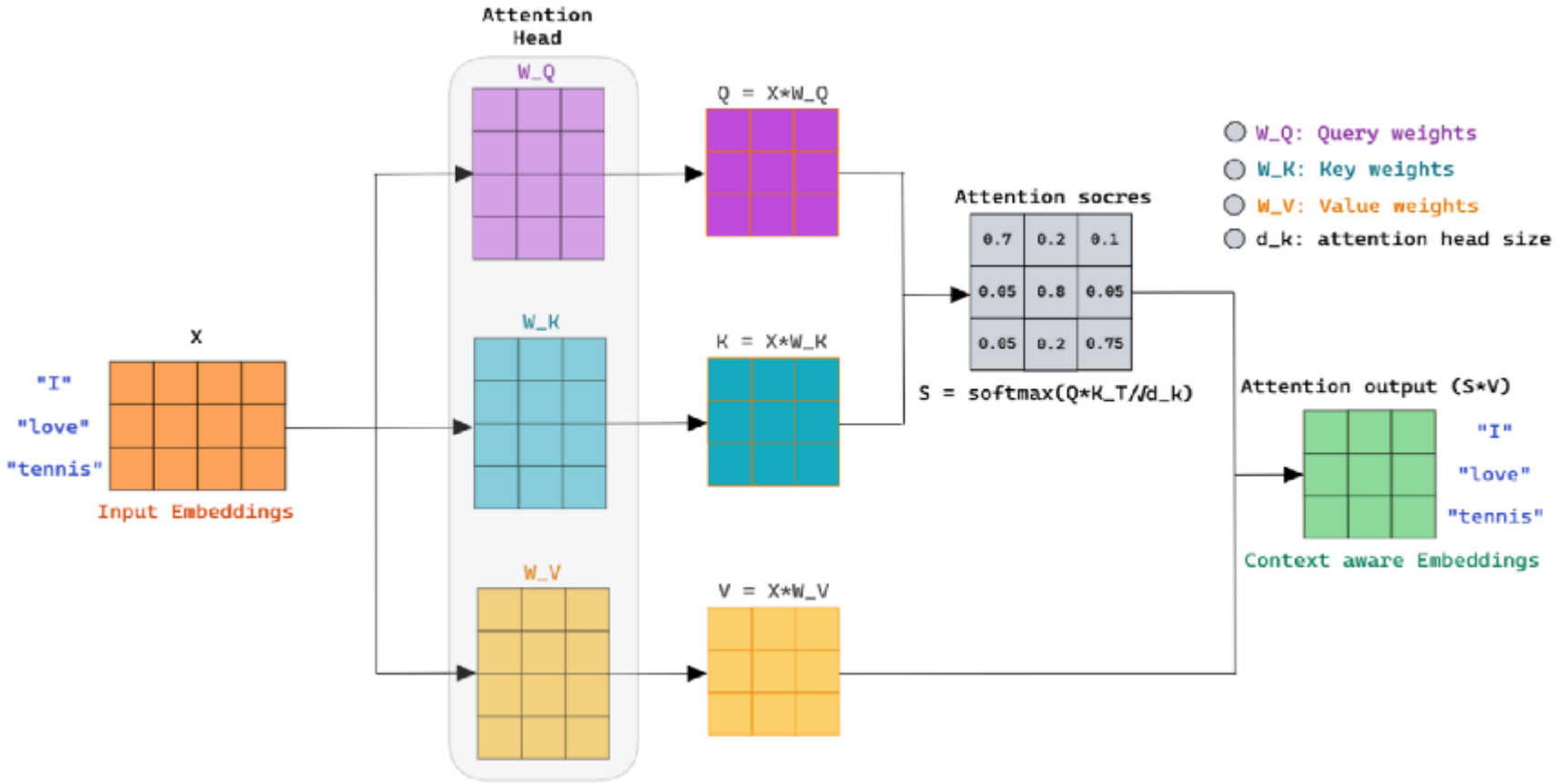
The diagram shows which input tokens a specific token is attending to at a given layer and head

Each colored line represents the strength of attention from the selected token (e.g., "it_") to another token in the sequence

Stronger, more saturated lines indicate higher attention weights, revealing where the model is focusing its contextual understanding

Attention Mechanism

- The diagram shows how AI **pays attention** in a sentence.
- The orange box is the input words (“I love tennis”).
- The middle part is the machine deciding **how strongly each word should pay attention to the others**.
- The green box on the right is the same words, but now each one is **updated with context** (e.g., “love” is tied to “tennis”).



https://www.youtube.com/watch?v=vsElpi4OZWc&list=PLyhJeMedQd9R7u-Dcgh1Zs-DjrAecEPm-

Context Aware Embeddings

Contextual Word Embeddings (BERT and GPT)

Key Innovation:

Unlike static embeddings Word2Vec, contextual models generate **different vectors for the same word** depending on its context.

Approach

- Uses deep, pre-trained neural networks (often transformer-based)
- Embeddings are derived from entire sentences, capturing syntax and semantics dynamically

Examples

- **BERT - Bidirectional Encoder Representations from Transformers (2018)**: Transformer-based neural networks trained with masked language modeling and next sentence prediction
- **GPT - Generative Pre-training Transformer (2018):** Transformer-based unidirectional language model focused on generation.

Characteristics

- Embeddings are **context-sensitive** (e.g., “bank” in “river bank” vs. “savings bank”)
- Each word is embedded based on its role in the sentence.
- Embeddings vary for the same word depending on its position and meaning.
- Significantly improve performance on downstream NLP tasks.

BERT: Bidirectional Encoder Representations from Transformers

- Developed by Google in 2018
- A **pre-trained language model** based on the **Transformer encoder**
- Reads text **bidirectionally**, enabling deep contextual understanding

Key Ideas

- Uses only the **encoder** stack of the Transformer
- Pre-trained on large text corpora, then fine-tuned on specific tasks

Key Advantages

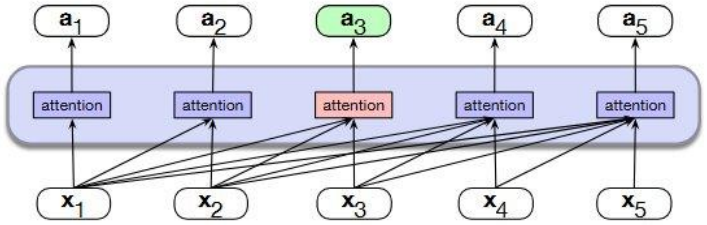
- Contextual embeddings: Word meanings change based on context
- Captures long-range dependencies
- State-of-the-art performance on 11 NLP tasks when released

BERT's Pre-Training Process

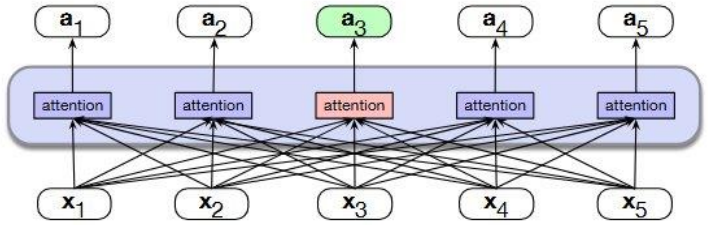
- **Training corpus**: 3+ billion words
  - English Wikipedia
  - ~10,000 unpublished books
- **Clean, large-scale dataset** for comprehensive language learning
- **Pre-training objective** to learn language patterns and context

Pretraining Objectives

- **Masked Language Modeling (MLM)**: Predict randomly masked words in a sentence
- **Next Sentence Prediction (NSP)**: Predict if one sentence follows another



a) A causal self-attention layer

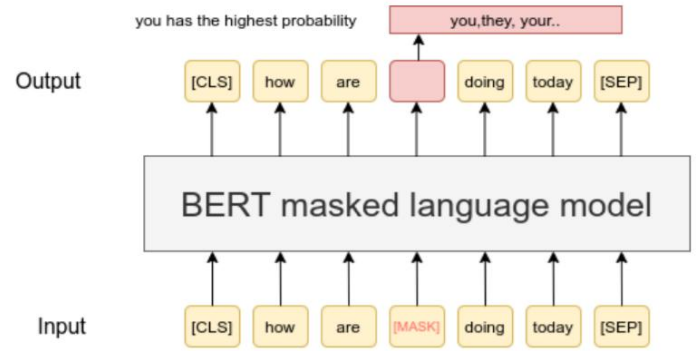


b) A bidirectional self-attention layer

Training Objective 1 - Masked Language Modeling

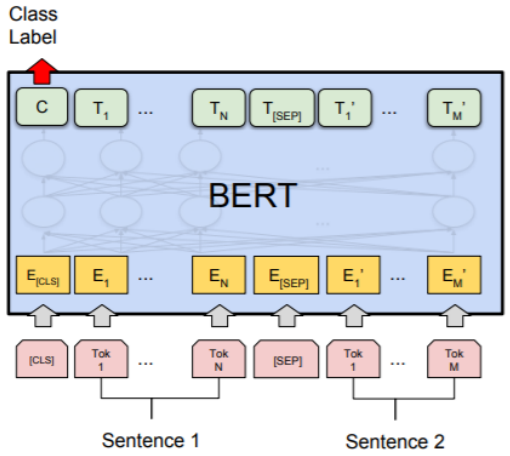
Learning Through "Fill in the Blanks"

- **Random word masking**: Some words are hidden during training
- **Context-based prediction**: BERT predicts masked words using surrounding context
- **Example**: "I love eating _____ in my fruit salad" → "lemons"
- **Key benefit**: Learns word relationships and contextual understanding
- **Powerful concept**: Used in many other AI applications



Training Objective 2 - Next Sentence Prediction

- **Title: Understanding Sentence Relationships**
- **Sentence pair analysis**: Given two sentences, determine if B follows A
- **Logical connection assessment**: Analyzes context and content
- **Example**:
  - A: "The cat climbed the tree"
  - B: "It was trying to catch a bird" ✓ (follows logically)
  - C: "The weather is nice today" ✗ (unrelated)
- **Note**: Later removed in newer models (MLM proved sufficient)



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

BERT's Ideal Use Cases

Where BERT Excels

- **Text Classification**: Sentiment analysis, topic classification, spam detection
- **Named Entity Recognition**: Identifying people, organizations, locations, dates
- **Extractive Question Answering**: Finding answers within provided context
- **Semantic Similarity**: Measuring similarity between sentences/paragraphs
- **Key advantage**: Perfect for understanding tasks, not generation

GPT – Overview and Architecture

What is Generative **P**re-trained **T**ransformer (GPT)?

- A family of **Transformer-based language models** developed by OpenAI
- Uses only the **decoder stack** of the original Transformer architecture
- Trained with **causal (autoregressive) language modeling** to predict the next token
- Focuses on generating human-like text (unlike BERT's understanding focus)

Training Objective

- Predict the next token in a sequence

GPT Variants

- **GPT-1 (2018, 120M Parameters)**: Introduced the pretrain-then-finetune paradigm
- **GPT-2 (2019, 1.5B Parameters)**: Scaled up model size, trained on web-scale data, Last publicly available weights
- **GPT-3 (2020, 175B Parameters)**: Enabled in-context learning, 100x boost, ~800GB file size
- **GPT-4 (2023, ~1T parameters (estimated))**: Multimodal, stronger reasoning and generalization

Applications (Wide range of NLP tasks through text generation)

- Text generation (e.g., chat, storytelling, code)
- Summarization
- Translation
- Question answering
- Semantic search and reasoning tasks

GPT's Architecture Deep Dive

How GPT Works

- **Multiple stacked decoder layers** with self-attention mechanisms
- **No encoders** - purely focused on text generation
- **Unidirectional processing** - considers only left-side context
- **Next word prediction** based on preceding words
- **High-quality text generation** from learned context patterns
- **Contextual understanding** combined with text creation capability

GPT Training Data

Pre-training Foundation

- **Massive, diverse datasets** including:
  - Web pages
  - Books and articles
  - Billions of words total
- **Different datasets** for each GPT version
- **Large-scale exposure** to language patterns and context
- **Quality varies** but emphasis on diversity and scale

Causal Language Modeling

GPT's Single Training Objective

- **Core task**: Predict the next word in a sequence
- **Method**: Uses context from preceding words only
- **Example**: "I love drinking fresh ___" → "lemonade"
- **Key benefits**:
  - Learns word relationships and context
  - Develops language patterns
  - Enables coherent text generation
- **Simplicity**: No masked language modeling or next sentence prediction

GPT's Ideal Use Cases

Where GPT Excels

- **Text Generation**: Story creation, creative writing, conversations
- **Instruction Following**: Responding to prompts and commands
- **Translation**: Converting text between languages
- **Summarization**: Creating concise summaries of longer texts
- **Code Generation**: Programming assistance and code completion
- **Versatile Applications**: Any task involving text output

Number of Parameters – GPT 3

Component	Parameter	Formula / Size	Total Parameters (Approx.)
Embedding Layer	Token Embeddings	Vocab × d_model = 50K × 12288	614.4M
	Positional Embeddings	n_ctx × d_model = 2048 × 12288	25.2M
Self-Attention (per layer)	Q, K, V, Output	4 × d_model × d_model = 4 × 12288^2	604.6M per layer
Feedforward (per layer)	2 layers (gelu)	d_model × d_ff + d_ff × d_model	1.2B per layer
LayerNorms (per layer)	Two per layer	2 × d_model	24.6K per layer
Total per layer	–	Self-attn + FFN + norms	≈ 1.8B per layer
Transformer Block Total	96 × 1.8B	–	≈ 172.8B
Final LayerNorm	–	d_model	12.3K
Output Layer (tied)	Shared with token embedding	–	— (tied with input embedding)
Total Parameters	–	Sum of above	**≈ 175B**

- **d_model = 12288**
- **n_layers = 96**
- **n_heads = 96 → each head has d_k = d_v = 128**
- **d_ff = 4 × d_model = 49152**
- **Vocabulary size ≈ 50,000**
- **Sequence length (n_ctx) = 2048**

GPT vs BERT Architecture

Aspect	GPT	BERT
Architecture	Decoder-only	Encoder-only
Text Processing	Unidirectional (left-to-right)	Bidirectional
Primary Goal	Text generation	Text understanding
Context	Previous words only	All surrounding words
Use Cases	Generation tasks	Classification/extraction

What is Hugging Face? 🤔

- **A company and a community platform** focused on democratizing Artificial Intelligence, especially Natural Language Processing (NLP) and Machine Learning (ML).
- Often called the "**GitHub for Machine Learning**."
- **Mission**: To make state-of-the-art ML models, datasets, and tools accessible to everyone.
- Started in 2016, initially with a chatbot app, then pivoted to open-source ML.

What does hugging face provide?

1. **Accessibility**: Provides easy access to thousands of pre-trained LLMs.
2. **Standardization**: Offers standardized tools and interfaces for working with different models.
3. **Collaboration**: Fosters a vibrant community for sharing models, datasets, and knowledge.
4. **Innovation**: Accelerates research and development in the LLM field.
5. **Ease of Use**: Simplifies complex ML workflows, from data preparation to model deployment.

Core Components of the Hugging Face Ecosystem

- **Hugging Face Hub**:
  - The central place to find, share, and collaborate on models, datasets, and ML applications (Spaces).
  - Over 1.7 million models, 75,000+ datasets!
- **Transformers Library**:
  - Python library providing thousands of pre-trained models for NLP, Computer Vision, Audio, and more.
  - Supports PyTorch, TensorFlow, and JAX.
  - Makes downloading, training, and using state-of-the-art models incredibly simple.
- **Datasets Library**:
  - Efficiently load and process large datasets.
  - Optimized for speed and memory, built on Apache Arrow.
  - Access to a vast collection of public datasets.
- **Tokenizers Library**:
  - Provides high-performance tokenizers crucial for preparing text data for LLMs.
  - Offers various tokenization algorithms and pre-trained tokenizers.

The Model Hub

Over 1,700,000+ Models Available

Popular Model Categories:

- **Text Generation**: GPT, LLaMA, Mistral, CodeLlama
- **Text Classification**: BERT, RoBERTa, DeBERTa
- **Question Answering**: BERT-based models
- **Translation**: T5, mT5, NLLB
- **Code Generation**: CodeT5, StarCoder
- **Multimodal**: CLIP, BLIP, LLaVA

Getting Started with Hugging Face

- Explore the Hub: huggingface.co
- Browse models, datasets, and Spaces.
- Install Libraries:

```
pip install transformers datasets tokenizers accelerate gradio
```

- Try a Pipeline:

```
# Example: Sentiment Analysis
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
result = classifier("Hugging Face is awesome!")
print(result)
# Example: Text Generation
generator = pipeline("text-generation")
output = generator("In a world of large language models,",
  max_length=50)
print(output)
```

Under the Hood

1. Automatic model selection
2. Tokenization handled
3. Inference optimization
4. Result formatting
5. Device management

Traditional Approach

1. Load tokenizer
2. Preprocess text
3. Load model
4. Run inference
5. Post-process results

... 50+ lines of code

Other Hugging face Pipelines

The Hugging Face `transformers` library supports a wide range of **pipelines**, each designed for a specific **natural language processing (NLP)** or **vision** task — so you can use powerful models without deep setup.

Pipeline Name	Task Description
"sentiment-analysis"	Classify sentiment (positive/negative)
"text-classification"	General text classification (multi-label or multi-class)
"zero-shot-classification"	Classify into labels **without training** on them
"text-generation"	Generate text (e.g., GPT models)
"text2text-generation"	Text-to-text tasks (e.g., summarization, translation)
"translation"	Translate between languages
"summarization"	Generate a summary of input text
"question-answering"	Extract answer from context
"fill-mask"	Predict missing word in a sentence (BERT-style)
"ner" (Named Entity Recognition)	Extract entities (like names, places, etc.)
"conversational"	Chatbot-style conversation
"sentence-similarity"	Measure similarity between two sentences
"token-classification"	Classify each token (used for NER, POS tagging, etc.)
"feature-extraction"	Extract embeddings/features from a model
"table-question-answering"	QA over structured data (tables)

➤ **Sentiment Analysis**

```python
pipeline("sentiment-analysis")("I love this!")
```

➤ **Summarization**

```python
pipeline("summarization")("Long article text goes here...")
```

➤ **Translation**

```python
pipeline("translation_en_to_fr")("This is amazing.")
```

➤ **Question Answering**

```python
qa = pipeline("question-answering")
qa({
    "question": "Where do pandas live?",
    "context": "Pandas are native to China and prefer bamboo forests."
})
```

To list all available pipelines in code:

```python
from transformers.pipelines import SUPPORTED_TASKS
print(SUPPORTED_TASKS.keys())
```

Open in Colab

Fine-Tuning Large Language Models

What is Fine-Tuning?

Fine-tuning is the process of further training a pre-trained language model on a specific dataset to adapt it for particular tasks, domains, or behaviors.

Unlike prompt engineering, fine-tuning modifies the model's parameters through additional gradient updates.

Why Fine-Tune Instead of Relying on Prompts?

- Prompts have limits in consistency and reasoning depth.
- Fine-tuning creates predictable outputs aligned with organizational requirements.
- Reduces lengthy prompting and improves inference quality.
- Enables use of proprietary or domain-specific data.

Types of Fine-Tuning

Full Fine-Tuning:

- Updates all model parameters
- Requires significant computational resources (GPUs/TPUs)
- Dataset size: Typically 10,000+ examples
- Risk of catastrophic forgetting

Parameter-Efficient Fine-Tuning (PEFT):

- LoRA (Low-Rank Adaptation): Adds trainable rank decomposition matrices
- Prefix Tuning: Optimizes continuous prompts in embedding space
- Adapter Layers: Inserts small trainable modules between layers
- Requires 0.1-1% of parameters compared to full fine-tuning

Instruction Fine-Tuning:

- Trains models to follow instructions better
- Uses dataset of (instruction, input, output) triplets
- Examples: FLAN, InstructGPT, Alpaca

Fine-Tuning Process

1. Data Preparation: Collect domain-specific or task-specific examples
2. Data Formatting: Structure as prompt-completion pairs
3. Training Configuration: Set learning rate (typically 1e-5 to 5e-5), batch size, epochs
4. Training: 3-10 epochs typical for most tasks
5. Evaluation: Monitor for overfitting using validation set

Fine-Tuning vs Prompt Engineering

When to Use Prompt Engineering:

Advantages:

- No training required - immediate deployment
- No computational costs
- Preserves model's general capabilities
- Rapid iteration and testing
- Works with API-only access

Best for:

- Prototyping and experimentation
- Few-shot learning scenarios (< 100 examples)
- General tasks with clear instructions
- When model updates frequently

When to Use Fine-Tuning

Advantages:

- Superior performance on specific tasks (10-30% improvement typical)
- Consistent output format and style
- Reduced inference costs (shorter prompts needed)
- Can learn new knowledge not in pre-training

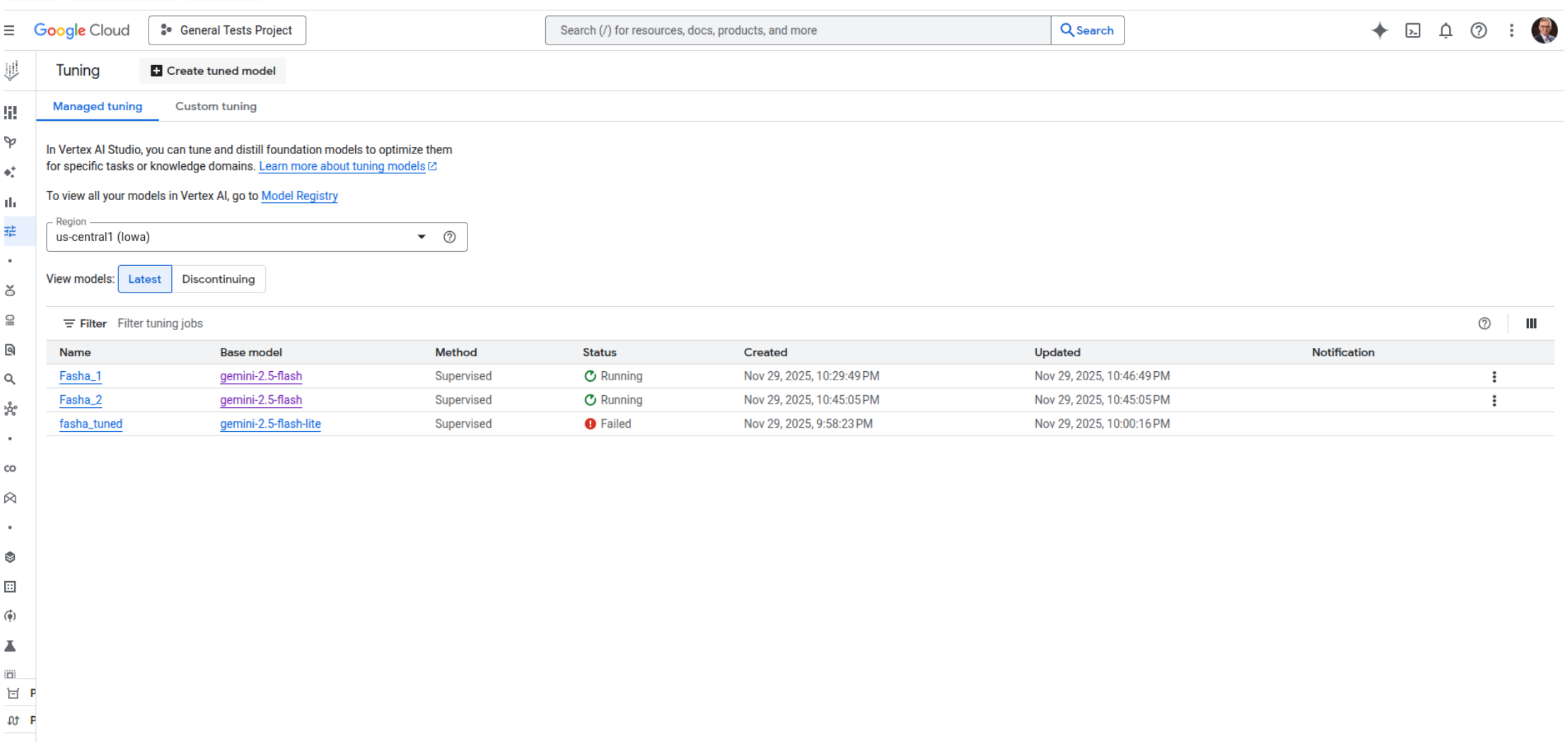
Best for:

- Domain-specific applications (medical, legal, technical)
- High-volume production use cases
- When you have 1000+ high-quality examples
- Custom formatting requirements

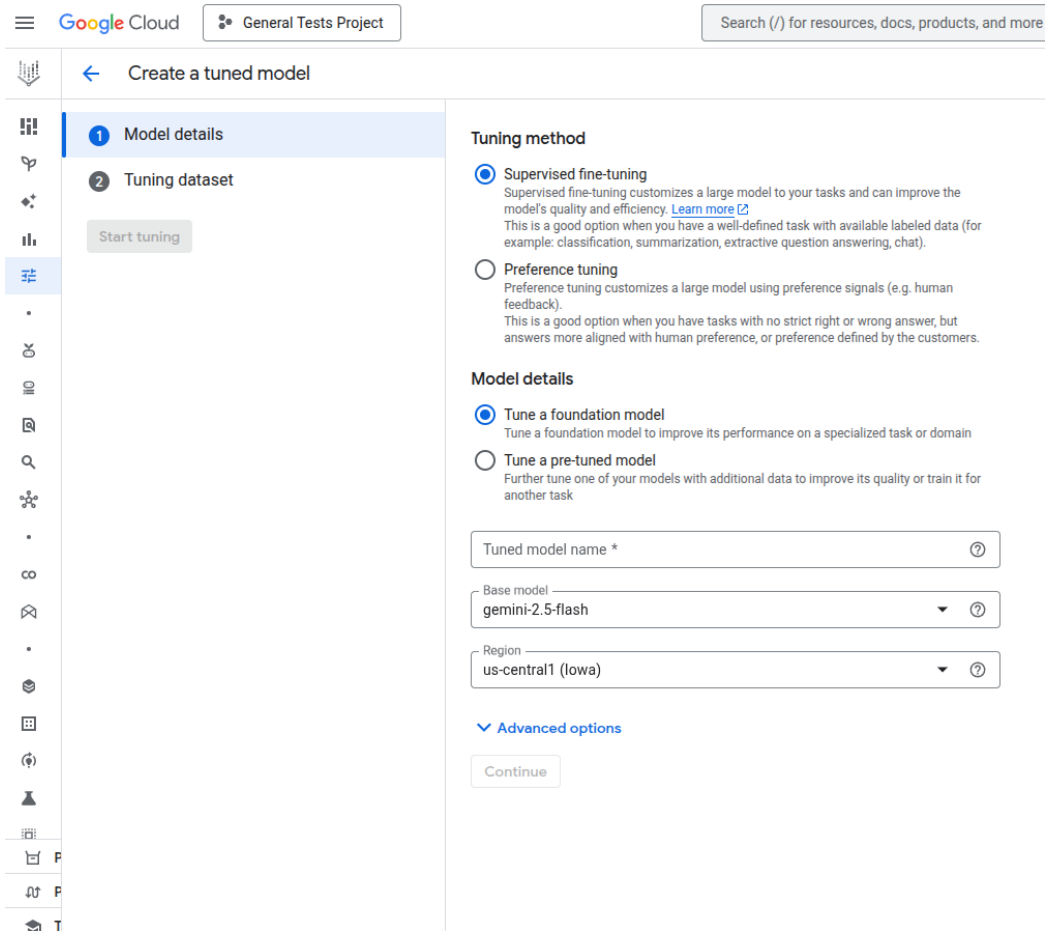
Fine-Tuning a Model on Vertex AI

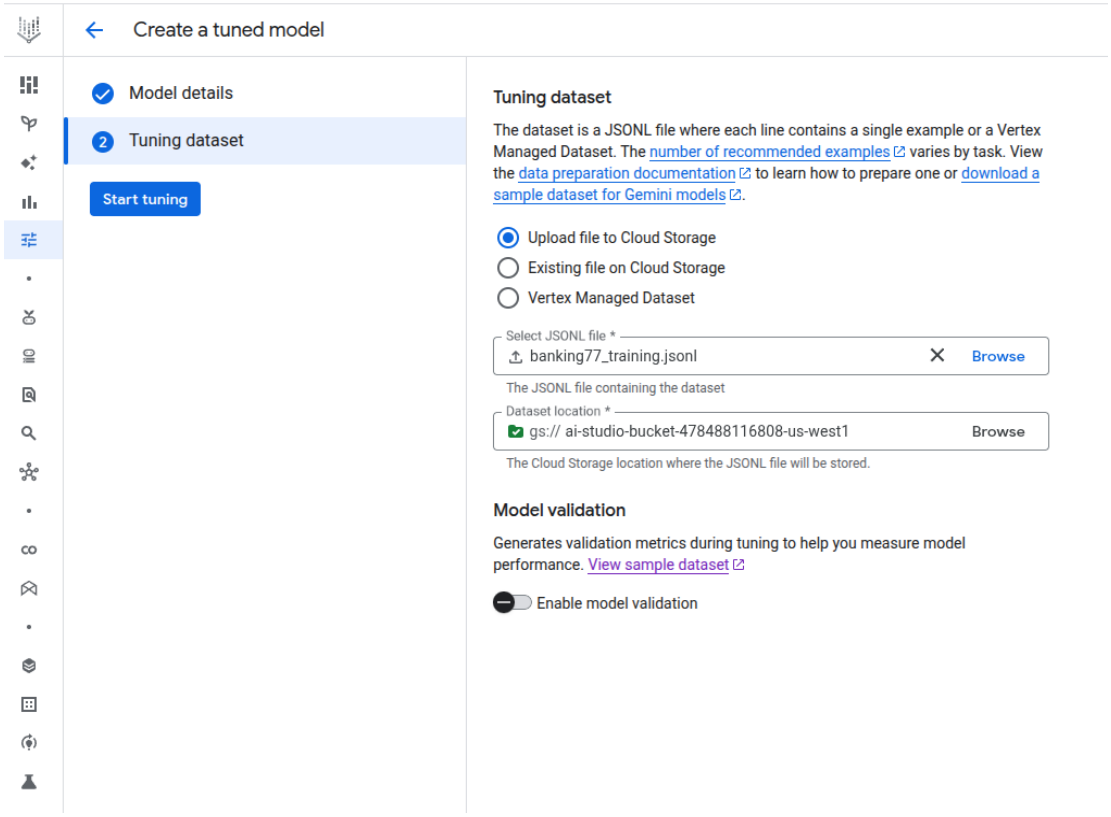
- Prepare a JSONL dataset with fields: instruction, input, output. (Prepare supervised fine-tuning data for Gemini models, Tune Gemini models by using supervised fine-tuning)
- Sample datasets can be found at: https://github.com/msfasha/307307-BI-Methods-Generative-AI/20251/vertexai_fine_tunning_data
- Save the file to your hard disk, for example: banking77_training.jsonl.
- Open Google Cloud Console and go to Vertex AI.
- Click “Tune a model” or “Fine-tune model”.
- Select a suitable base model such as Gemini 2.5 flash-lite.
- Upload your JSONL dataset when prompted.
- Name the tuned model, for example: banking-intent-classifier-v1.
- Start the tuning job and wait for it to finish.

Click on Create a Tuned Model in VertexAI



Select the Base Model and Select The Training Dataset





Testing and Evaluating the Tuned Model

- After training completes, open the tuned model in Generative AI Studio.
- Use the Test panel to enter new banking queries.
- Compare responses from the tuned model and the base model.
- Confirm that outputs follow your desired format (for example: intent: chargeback).
- Check edge cases such as ambiguous or short inputs.
- Identify misclassifications and note what examples should be added.
- Update your dataset and prepare for future tuning iteratio

Deploying and Using the Tuned Model

- From the tuned model page, click “Deploy to endpoint” if integration is required.
- Create a new endpoint or reuse an existing one.
- Assign minimal compute for testing and scale later if needed.
- Call the tuned model through the Vertex AI API, web UI, or application backend.
- Monitor output quality over time and periodically refresh the dataset.
- Repeat fine-tuning when new banking intents or patterns appear.
- Store multiple versions of the tuned model for rollback and comparison.